AD-A045 876 NAVAL SURFACE WEAPONS CENTER DAHLGREN LAB VA F/G 9/2
AN INFORMAL INVESTIGATION INTO SOFTWARE ENGINEERING AS APPLIED --ETC(U)
APR 77 J G PERRY
UNCLASSIFIED NSWC/DL-TR-3625 NL

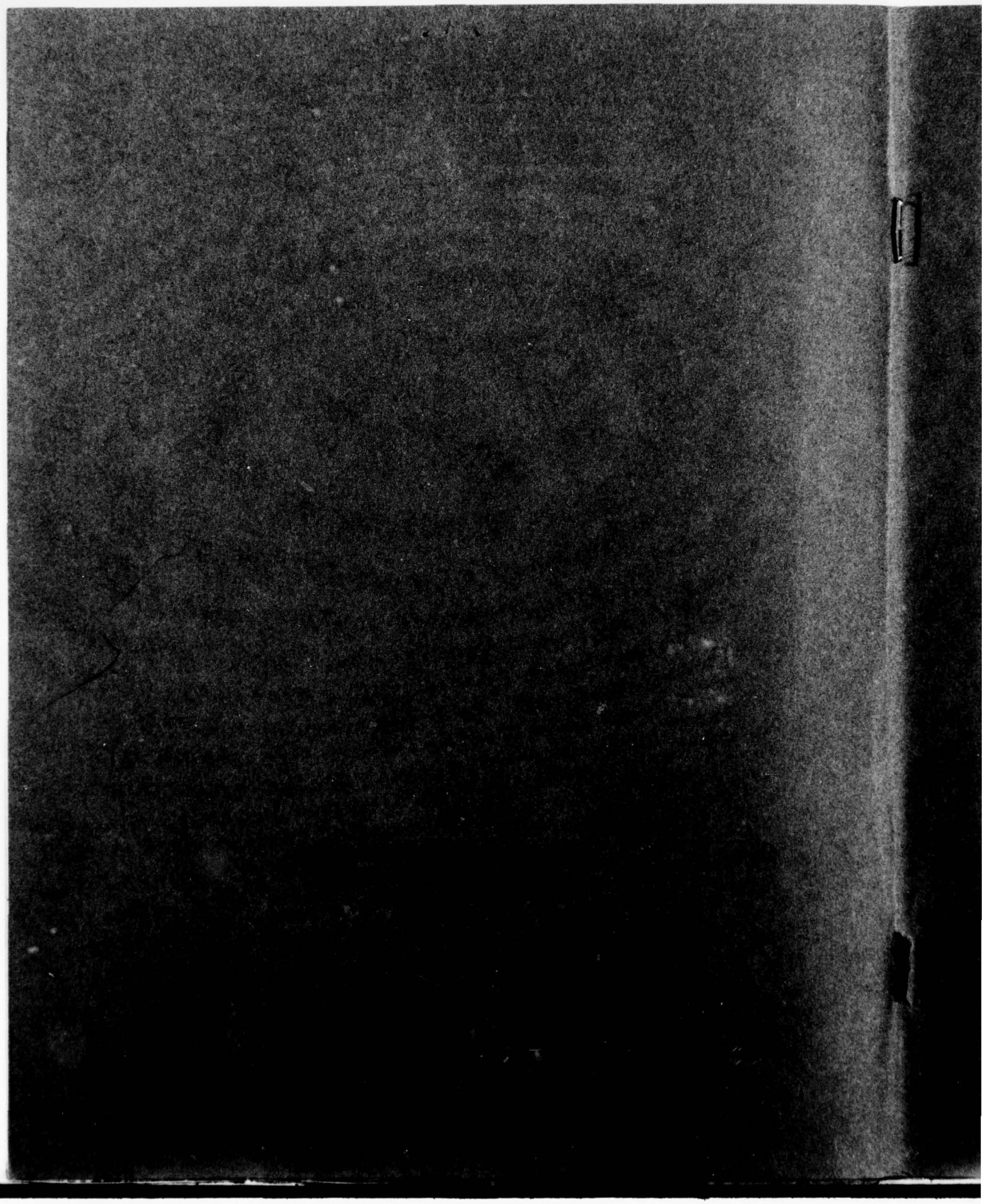

END
DATE FILMED
11-77
DDC

NSWC/DL TR-3625

# AN INFORMAL INVESTIGATION INTO SOFTWARE ENGINEERING AS APPLIED TO A COMPILER BOOTSTRAPPING PROJECT

by
JOHN G. PERRY, JR.
Warfare Analysis Department

APRIL 1977



UNCLASSIFIED (14) NSWC/DL-TR-3625

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

## REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TR-3625
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): (6) AN INFORMAL INVESTIGATION INTO SOFTWARE ENGINEERING AS APPLIED TO A COMPILER BOOTSTRAPPING PROJECT.

5. TYPE OF REPORT & PERIOD COVERED: (9) Final rept.

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): (10) John G. Perry, Jr.

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Surface Weapons Center (DK-74)
Dahlgren Laboratory
Dahlgren, Virginia 22448

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
CDC 6700 Computer
Program Support

11. CONTROLLING OFFICE NAME AND ADDRESS

12. REPORT DATE: (11) April 1977

13. NUMBER OF PAGES: (12) 59p.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.



17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Bootstrapping
UNIVAC 1108 SIMPL-T Compiler
CDC 6700 Computer
Structured Programming
Modular Design
Software Engineering

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes in detail the necessary steps involved and the problem areas discovered in transporting (bootstrapping) the UNIVAC 1108 SIMPL-T compiler to a CDC 6700 computer system. SIMPL-T is an elementary procedure-oriented language that conforms to the standards of structured programming and modular design. Certain informal experiments relating to software engineering were conducted during this project. A brief summary of software engineering experiments and results follows.

(see back)

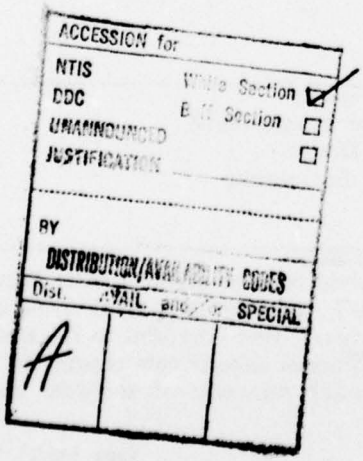
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

391598

(20)

A detailed log of the man-hours of work needed to complete this task was partitioned according to the following categories: design (22 percent), coding (23 percent), testing (45 percent), and other (10 percent). The initial manpower estimates were low by about 17 percent. Both the use of a "program description language" and a "high level programming language" appear to have improved programmer productivity. Also, error analysis indicated that the number of errors made per line of code for high level lines of code (SIMPL-T) and assembly lines of code (COMPASS) was about the same.



## FOREWORD

The purpose of this project was threefold: (1) to increase the author's knowledge of compilers and compiler writing; (2) to look at and examine the problems involved in bootstrapping a transportable compiler from one system to another system; and (3) to investigate the advantages of using "software engineering concepts" over traditional approaches.

The project was completed while the author was participating in the Long-Term Training Program at the Naval Surface Weapons Center/Dahlgren Laboratory (NSWC/DL). Computer time was provided by the University of Maryland Computer Center and also by the Computer Facility Division at NSWC/DL.

This report was reviewed by Hermon W. Thombs, Head, Program Systems Branch, Computer Program Division, and W. P. Warner, Head, Computer Program Division.

Released by:

Ralph A. Niemann

RALPH A. NIEMANN, Head
Warfare Analysis Department

## ACKNOWLEDGEMENTS

I would like to thank Dr. Victor R. Basili, Associate Professor of Computer Science at the University of Maryland, for guidance on the overall project and Mr. Albert J. Turner, candidate for a Doctor of Philosophy in Computer Science at the University of Maryland, for his help with the UNIVAC 1108 SIMPL-T compiler and UNIVAC system.

## TABLE OF CONTENTS

## LIST OF ILLUSTRATIONS



## LIST OF TABLES

## INTRODUCTION

Two approaches to compiler bootstrapping are presented: classical and SNOBOL. The project environment and technical background information provided allows the reader to be better acquainted with the project.

## BOOTSTRAPPING TECHNIQUE

Two approaches to bootstrapping as used to transport a compiler from one machine to another are discussed below.

### Classical Approach

In a classical approach, the code generator of the host computer's compiler is modified to produce code (e.g., assembly language or a higher order language) to be executed by the target computer. The specific code generated by the host is itself a compiler to be run on the target machine.

In general, a transportable compiler in the host machine is written in the language of the compiler (e.g., the AED compiler is written in the AED language;[1] the XPL compiler is written in the XPL language;[2] and the SIMPL-T compiler is written in SIMPL-T language.[3]) However, there are necessary interface routines with the operating system which are not part of the compiler, but are essential to the use of the compiler. These interface routines (also called "run-time" routines or "language support" routines) are written in the assembly language of the host machine. Therefore, the transporting of a compiler from the host machine to the target machine requires a twofold installation. Not only must the compiler itself be transferred to the target machine (via modification to the code generator of the host machine's compiler as described above), but the interface routines (e.g., input-output, conversion, and string handling routines which normally complement the basic language) must be written for the target machine as an essential supplement to the basic compiler.

To implement the system described, the compiler generated by the host machine (in a target machine language) is actually installed on the target machine with its accompanying interface routines. As a check, the compiler (written in the language of the compiler) is compiled for the first time on the target machine's compiler. The resulting output, in a target machine language, should correspond one

to one with the compiler as it was generated by the host in the same target machine language (for implementation on the target machine). Diagram 1 presents the classical approach to compiler bootstrapping.

Assuming successful testing, a self-contained compiler is running on the target machine, with no further dependence upon the host machine.

**Diagram 1. Classical Approach**

| GIVEN | GOAL |
|---|---|
| $C_H^H [S^H] \Rightarrow C_H^H$ | $C_T^T [S^T] \Rightarrow C_T^T$ |
| Compiler (C) in machine language, running on the host machine (superscript H) and generated on the host machine (subscript H), is given a source (S) as input. The source (S) is a compiler written in its own language that generates machine language for the host computer (superscript H).<br><br>The result, $C_H^H$, is a compiler (C) written in machine language, that runs on the host machine (superscript H) and is generated on the host machine (subscript H). | Compiler (C) in machine language, running on the target machine (superscript T) and generated on the target machine (subscript T) is given a source (S) as input. The source (S) is a compiler written in its own language that generates machine language for the target computer (superscript T). The result, $C_T^T$, is a compiler (C) written in machine language, that runs on the target machine (superscript T) and is generated on the target machine (subscript T). |

Where

C = compiler in machine language [When C has a subscript, the subscript indicates the machine that generated the compiler (C); and when C has a superscript, the superscript indicates the machine that the compiler (C) runs on.]
H = host computer
S = source: the input, the compiler written in the compiler's own language [When S has a superscript, the superscript indicates the machine that the source compiler (S) generates code for.]
T = target computer

Diagram 1. Classical Approach (Continued)

| PROCEDURE | COMMENT |
|---|---|
| $C_H^H [S^H] \Rightarrow C_H^H$ | (given) |
| $C_H^H [S^T] \Rightarrow C_H^T$ | Input to the host's compiler is a source (S) written to generate machine language for the target computer (superscript T). The result is the compiler in machine language (C) generated by the host computer (subscript H) to run on the target computer (superscript T). |
| $C_H^T [S^T] \Rightarrow C_T^T$ | The compiler created above ($C_H^T$) to run on the target machine (superscript T) is run on the target machine, with a source ($S^T$) as input that generates code for the target machine. The result ($C_T^T$) is a compiler (C) written in machine language, generated by the target machine (subscript T) to run on the target machine (superscript T). |
| $C_T^T [S^T] \Rightarrow C_T^T$ | The compiler generated above ($C_T^T$) receives $S^T$ as input and yields $C_T^T$. |

**SNOBOL Approach**

The major advantage of this approach over the Classical Approach is that it does not require the use of the host computer. Assume the existence of a standard SNOBOL program that will convert a SIMPL-T program to a standard FORTRAN program, both the standard SNOBOL program (functioning as a SIMPL-T compiler) and the FORTRAN program produced would be capable of running on either the target or the host machine. Therefore, the SIMPL-T compiler could be bootstrapped onto a target machine using only the target machine.[4]

The procedure for compiler boostrapping is accomplished in small steps, with the first two steps being the same as for the classical approach:

1. Modify the code generator of the host computer's compiler to produce code for the target machine.

2. Write the necessary run time interface routines for the target machine.

3. Compile the modified compiler (modified to produce code for the target machine) on the target machine using the SNOBOL program which converts SIMPL-T programs to FORTRAN. The resulting FORTRAN program performs the same function as the original modified SIMPL-T compiler; namely, the FORTRAN program is a SIMPL-T compiler, accepting SIMPL-T as input and yielding a machine language for the target machine.

4. Compile the modified SIMPL-T compiler (the SIMPL-T compiler written in SIMPL-T, but modified to produce code for the target computer) on the target computer using the SIMPL-T compiler (generated in FORTRAN). This process yields a SIMPL-T compiler for the target computer that produces machine language for the target computer.

Diagrams 2 and 3 illustrate the procedure explained above.

**Diagram 2. SNOBOL Approach**

| GIVEN | GOAL |
|---|---|
| $B^T$<br>(Note: the host machine is not used.) | $C_T^T [S^T] \Rightarrow C_T^T$ |

Where

$B^T$ = standard SNOBOL program run on the target machine that converts a SIMPL-T program(s) to an equivalent FORTRAN program (F)

$S^T$ = SIMPL-T compiler written in SIMPL-T to generate code for the target machine (superscript T)

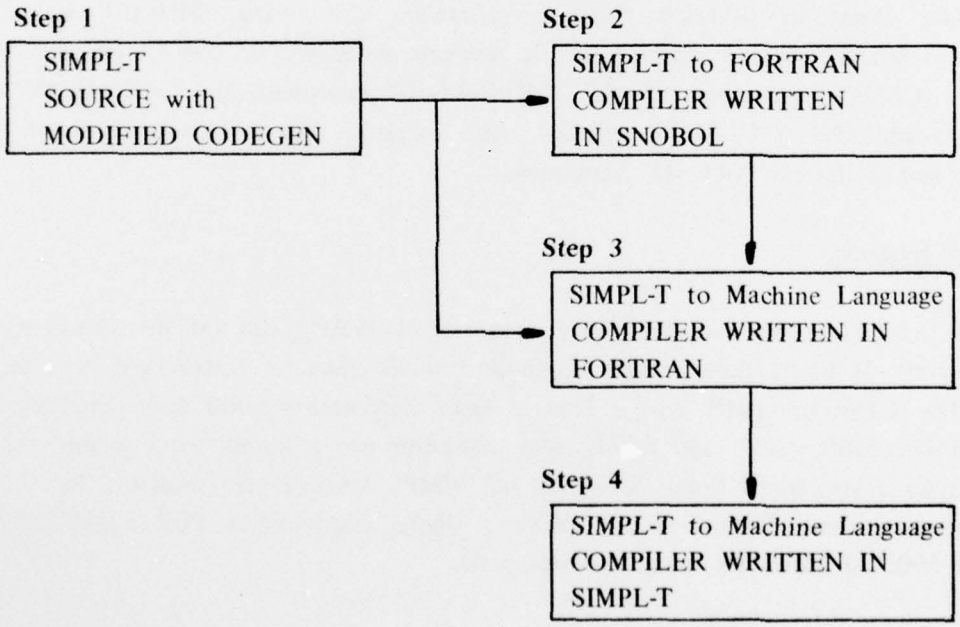
$F_T^T$ = standard FORTRAN program generated by the target machine to run on the target machine that acts as a compiler in converting a SIMPL-T program to machine language

$C_T^T$ = SIMPL-T compiler written in machine language by the target machine to run on the target machine

Diagram 2. SNOBOL Approach (Continued)

| PROCEDURE | COMMENT |
|---|---|
| $B^T [S^T] \Rightarrow F_T^T$ | Input for the SNOBOL program ($B^T$) is the SIMPL-T compiler written in SIMPL-T ($S^T$). The result is the same SIMPL-T compiler written in FORTRAN ($F_T^T$). |
| $F_T^T [S^T] \Rightarrow C_T^T$ | Input for the FORTRAN SIMPL-T compiler ($F_T^T$) is the SIMPL-T compiler written in SIMPL-T ($S^T$). The result is the SIMPL-T compiler written in machine language for the target computer ($C_T^T$). |
| $C_T^T [S^T] \Rightarrow C_T^T$ | The compiler generated above receives $S^T$ as input and yields $C_T^T$. |

Diagram 3. SNOBOL Approach: Flow Diagram



## BACKGROUND

### SIMPL Family[5]

SIMPL-T is just one of several languages which belong to a family. A family is a set of languages with a common base. The SIMPL family is united on the basis of a common set of language features, including control and data structures, scoping rules, etc. Separate languages within the family facilitate the solving of problems in certain areas. For example, there is a system version of SIMPL (SIMPL-S[6]) and there is a real arithmetic version of SIMPL (SIMPL-R[7]). A graphic version of SIMPL (SIMPL-G[8]) is planned.

The SIMPL-T language is the transportable version of the family-designed to be relatively machine independent, with compilers that are relatively transportable to a variety of machines. It is a procedure-oriented language that was designed to conform to the standards of structured programming and modular design. There are three data types in SIMPL-T: integer, character, and string. SIMPL-T is an elementary, straight-forward language which supports recursion. Its control statements are: the "CASE" statement, the "IF...THEN...ELSE" statement, the "WHILE...DO" statement, and the ""CALL" statement. The language has a three-level scoping structure and it has no "GO TO" statements.

### Computer Systems

The host computer for this project was the UNIVAC 1108 (at the University of Maryland). It has a 36-bit word length and some character instructions (quarter word). The target computer was a Control Data Corporation 6000 series computer (6200, 6400, 6500, 6600, and 6700). The computer has a 60-bit word length and no character instructions. Some forms of the SIMPL language are available for the following computers: Data General's NOVA, Digital Equipment's PDP 11/45, and the IBM 360/370 (currently being bootstrapped).

### Environment

There was only a minimal amount of documentation available on the SIMPL-T compiler. The host computer (UNIVAC 1108) is located approximately 70 miles from the target computer (CDC 6700). Interactive computing was used on both machines whenever possible.

### Estimations

Some initial estimations concerning projected time expenditures were made before the project was undertaken. The original estimates were as follows:

| | |
|---|---|
| PHASE 1 – UNIVAC 1108 | |
| Background and Design | 1 mo |
| New SIMPL-T code generator | 1 mo |
| PHASE 2 – CDC 6700 | |
| CDC SIMPL-T runtime routines/ functions | 1 mo |
| Testing | 1 mo |
| Total estimated work on projects | 4 mo |

## PROJECT

## BOOTSTRAPPING PROCEDURE

### SNOBOL Approach Abandoned

For this experiment, the SNOBOL program was not used; although, a SNOBOL program exists that converts a SIMPL-T program to a FORTRAN program. The necessary run-time support routines were already available in FORTRAN, and the

SNOBOL approach was used for the IBM 360/370 bootstrapping. However, it was not feasible to use SNOBOL in this study. An explanation for this decision is provided below:

1. The original SNOBOL system on the target machine (the CDC 6700) only supported 30-bit words and the SIMPL-T compiler required at least 32-bit integer arithmetic.

2. The SNOBOL conversion program needed several scratch files, but the SNOBOL system on the target machine only provided one.

The scratch file problem might have been corrected, but there was no obvious way to solve the inconsistency in word lengths; therefore, two other different versions of SNOBOL were requested from different universities.

The first alternate SNOBOL system received, supported the full 60-bit word but did not have all of the SNOBOL-4 functions used by the SNOBOL conversion program. In addition, the character set was different from the one in use on the target machine. Here, the character set problem could be corrected, but the missing functions in this version of the SNOBOL system made it unusable.

After a considerable time delay, the second SNOBOL system arrived. This SNOBOL system had support for a 36-bit word and all the necessary SNOBOL-4 functions. However, there was a character set difference, and the added task of installing the SNOBOL compiler on the target computer seemed unnecessary in the light of current developments. Were there no other alternatives, this alternative could have been pursued.

Progress on the classical approach to bootstrapping was made during the considerable time delay (several months) that occurred while searching for an adequate SNOBOL system. The SNOBOL approach was therefore abandoned.

**Classical Approach – General Discussion**

The classical approach can be divided into two phases. The first phase is oriented toward the UNIVAC 1108 computer (the host machine). The code generator of the host computer's compiler was redesigned and reprogrammed to take into consideration the run-time environment of the target machine (CDC 6700). The decision to use CDC assembly language (COMPASS) as the target language was made to keep the project as uncomplicated as possible. This decision reduced the over-all

analyst/programmer time expenditure for the project, increased compiler flexibility, and the COMPASS code could be executed on different operating systems and/or could be changed more easily than binary. Due to the magnitude of this decision, a more detailed explanation of this choice of target language follows at the conclusion of this section.

The second phase of the classical approach is oriented to the target machine. The necessary run-time support routines (string packages, input/output, etc.) which enable the new compiler to integrate successfully with the target computer's operating system were written in COMPASS for the CDC machine. Actual installation of the new compiler (the SIMPL-T compiler produced by the host and written in COMPASS for the target machine) followed. After installation, the new compiler on the CDC 6700 was used to recompile the modified SIMPL-T compiler (written in SIMPL-T). Once tested, this compiler provides a closed system, separate and apart from the host machine. Any changes to the CDC version of SIMPL-T are made directly on the target machine.

The basic ways that the CDC compiler is restricted in comparison to its *UNIVAC 1108 version follow:*

1. All procedures and functions are recursive.

2. Case statement numeric selectors must be between 0 to 99.

3. The call statement arguments list cannot contain more than 20 parameters.

4. Most trace and statistical options are not supported.

5. The READ command is not supported. (The compiler does not make use of this command. This command will be incorporated at a later date.)

6. Both operands of a logical operator (.AND., .OR., ...) are always evaluated.

Returning to the topic of the selection of the target machine, CDC assembly language was chosen over machine language (relocatable binary) for reasons discussed below:

1. The generation of relocatable binary is cumbersome in the CDC system. The major reason for this difficulty lies in the fact that up to four machine instructions can exist within one 60-bit word. This density of instruction demands that the algorithms for relocating code and forward references be more sophisticated than they are for other large-scale machines.

2. The CDC 6700 operating system was, at the time, in the process of being converted over to a new operating system (from SCOPE 3.3 to SCOPE 3.4). When the project started, it was unsettled as to possible changes in relocatable format, character set, and input/output and system macro routines.

3. An important reason for producing assembly language was the distance between the two systems (host about 70 miles away from target machine). No tele-processing or tele-communication existed between the two computer sites. It was realized that it would be desirable to be able to make minor changes (i.e., a program error in the code generator) efficiently and quickly. This would be extremely difficult if relocatable binary code were being produced [i.e., not only would the error have to be corrected, but a corresponding adjustment in the error detection binary bit count (check sum) would probably have to be made].

4. Using assembly language eased testing of the code generator on the host computer. Visual checking had to be done of the code generated on the UNIVAC system. Symbolic code is more easily read than a binary representation.

5. This approach meant that two different internal character codes were no problem. If binary code had been used, the code generator would have to do additional code conversion. Since assembly language (COMPASS) was used, the checkout could be done entirely in the UNIVAC character set. Before the UNIVAC file was shipped to the CDC system, it would first be converted from the UNIVAC code to the CDC code.

The above reasons were the basis for selecting COMPASS as the target language. The primary consideration for the selection was the fact that COMPASS produced a functional system within a minimum amount of time.

The bootstrapping process required five versions of the SIMPL-T compiler:

1. The first version ran on the UNIVAC 1108 and produced COMPASS code for the CDC 6700 (i.e., crosscompiler).

2. The second version ran on the CDC 6700 and was not overlaid.

3. The third version was an overlaid version for the CDC 6700.

2. The CDC 6700 operating system was, at the time, in the process of being converted over to a new operating system (from SCOPE 3.3 to SCOPE 3.4). When the project started, it was unsettled as to possible changes in relocatable format, character set, and input/output and system macro routines.

3. An important reason for producing assembly language was the distance between the two systems (host about 70 miles away from target machine). No tele-processing or tele-communication existed between the two computer sites. It was realized that it would be desirable to be able to make minor changes (i.e., a program error in the code generator) efficiently and quickly. This would be extremely difficult if relocatable binary code were being produced [i.e., not only would the error have to be corrected, but a corresponding adjustment in the error detection binary bit count (check sum) would probably have to be made].

4. Using assembly language eased testing of the code generator on the host computer. Visual checking had to be done of the code generated on the UNIVAC system. Symbolic code is more easily read than a binary representation.

5. This approach meant that two different internal character codes were no problem. If binary code had been used, the code generator would have to do additional code conversion. Since assembly language (COMPASS) was used, the checkout could be done entirely in the UNIVAC character set. Before the UNIVAC file was shipped to the CDC system, it would first be converted from the UNIVAC code to the CDC code.

The above reasons were the basis for selecting COMPASS as the target language. The primary consideration for the selection was the fact that COMPASS produced a functional system within a minimum amount of time.

The bootstrapping process required five versions of the SIMPL-T compiler:

1. The first version ran on the UNIVAC 1108 and produced COMPASS code for the CDC 6700 (i.e., crosscompiler).

2. The second version ran on the CDC 6700 and was not overlaid.

3. The third version was an overlaid version for the CDC 6700.

4. The fourth version allowed constants of any size. (This version had no checking of size of constants; thus, allowing the bypass of the 36-bit word size difference.)

5. The fifth version was overlaid and supported 60-bit constants.

The following section deals with the development of these five versions.

## Classical Approach – Detailed Steps

Diagrams 4, 5, 6, and 7 show the basic bootstrapping steps used to move a SIMPL-T compiler written in SIMPL-T (and running on the UNIVAC 1108) to the CDC 6700. Five different compilers were generated. A description and reason for each step of the procedure is given below.

Diagrams 4, 5, 6, and 7 show the flow as each compiler generates the next. The diagrams have been organized in the following manner:

.0 Each page begins with an integer block that represents the new version of the SIMPL-T compiler, as it exists on the computer in its executable state (i.e., in machine language).

.1 Represents the source (a previously existing SIMPL-T compiler) written in SIMPL-T and used as input for the .0 boxes.

.2 Represents the COMPASS version of the changed SIMPL-T compiler – the output of feeding the contents of the .1 box through the .0 box.

.3 Represents the runtime routines (the user library) which allow interfacing with the operating system.

.4 Represents the installation process – the combining of the COMPASS version of the new compiler with the necessary runtime routines from the user library. The installation process always results in a new executable version of the SIMPL-T compiler.

**Diagram 4. SIMPL-T Compiler 2**

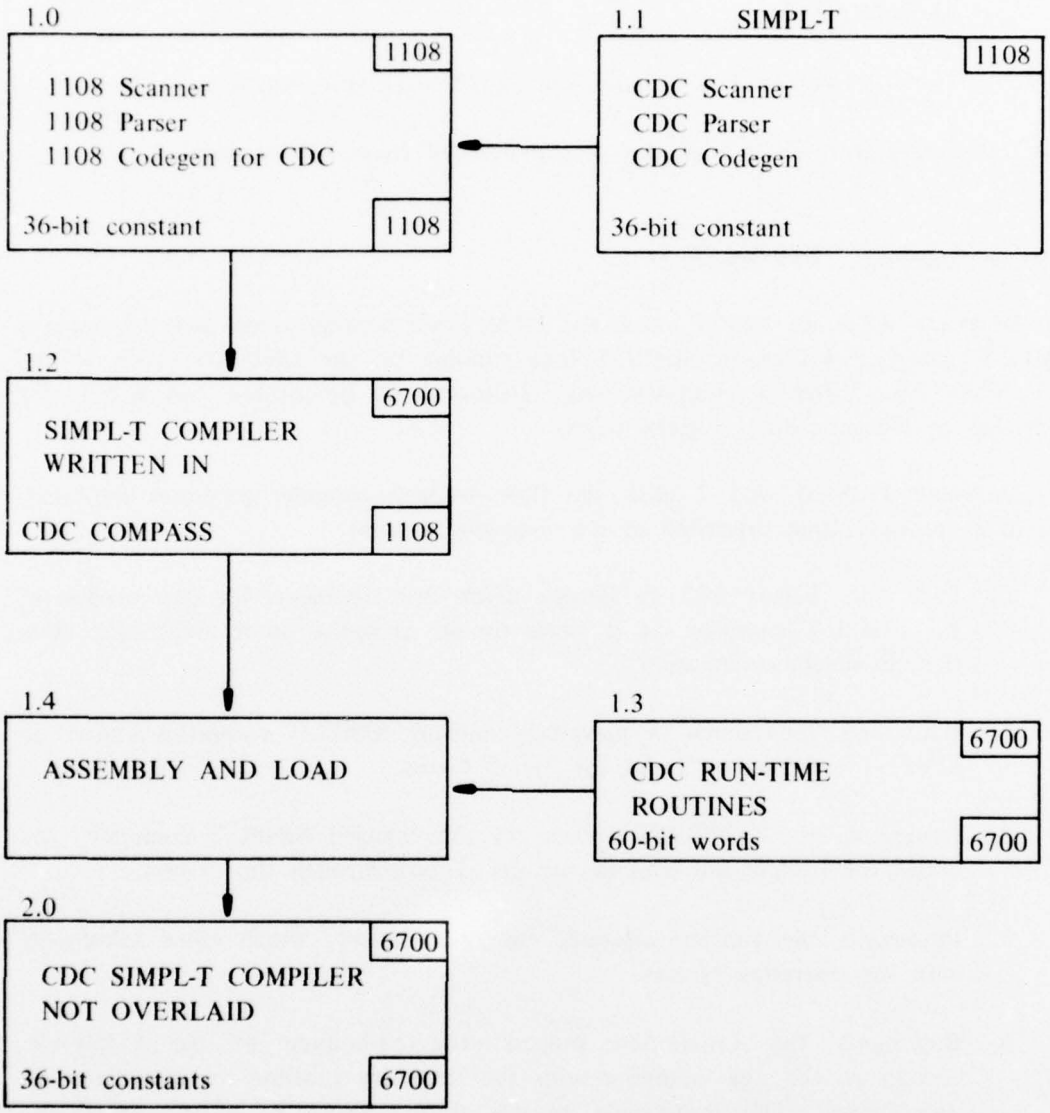


Diagram 5. SIMPL-T Compiler 3

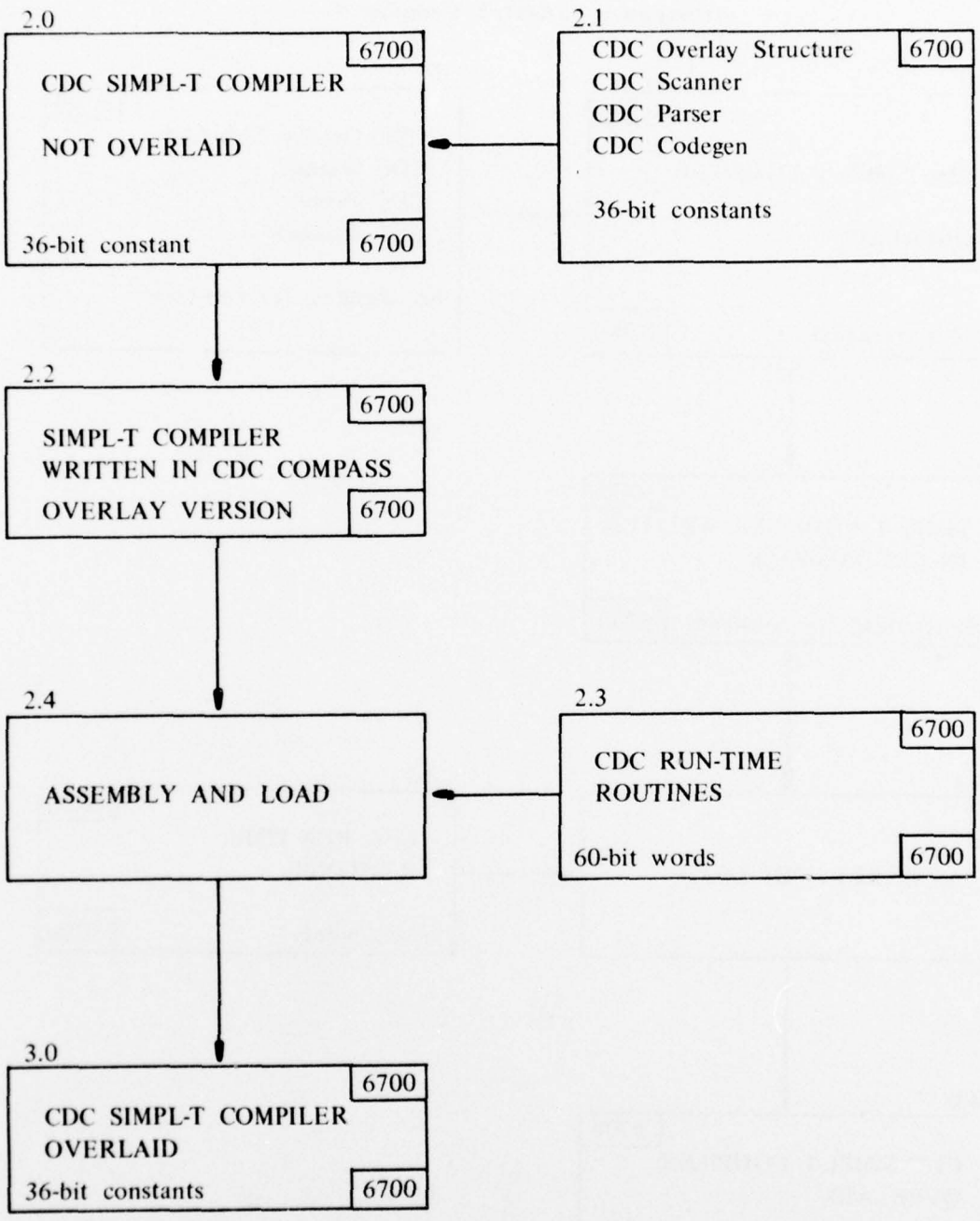


Diagram 6. SIMPL-T Compiler 4

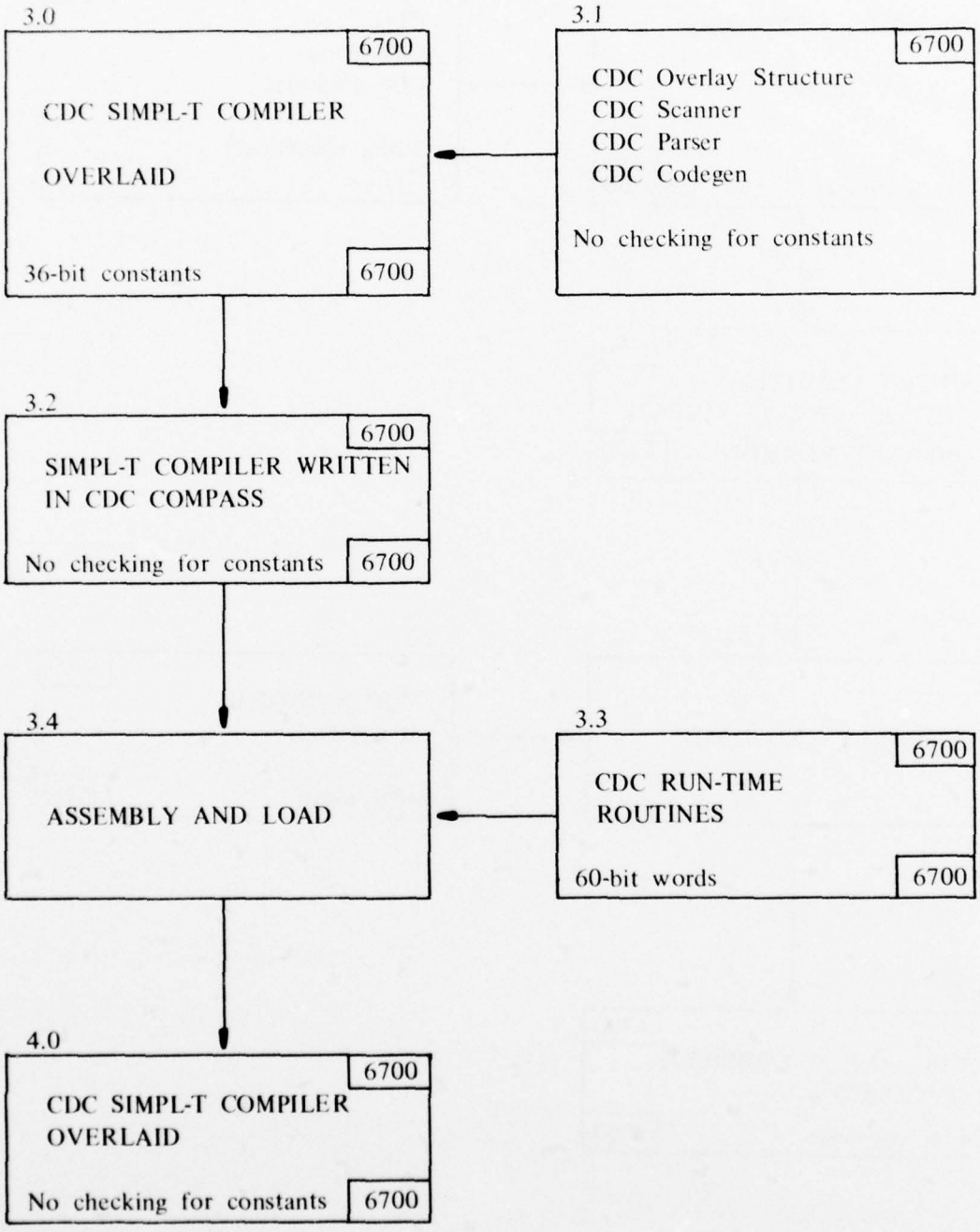
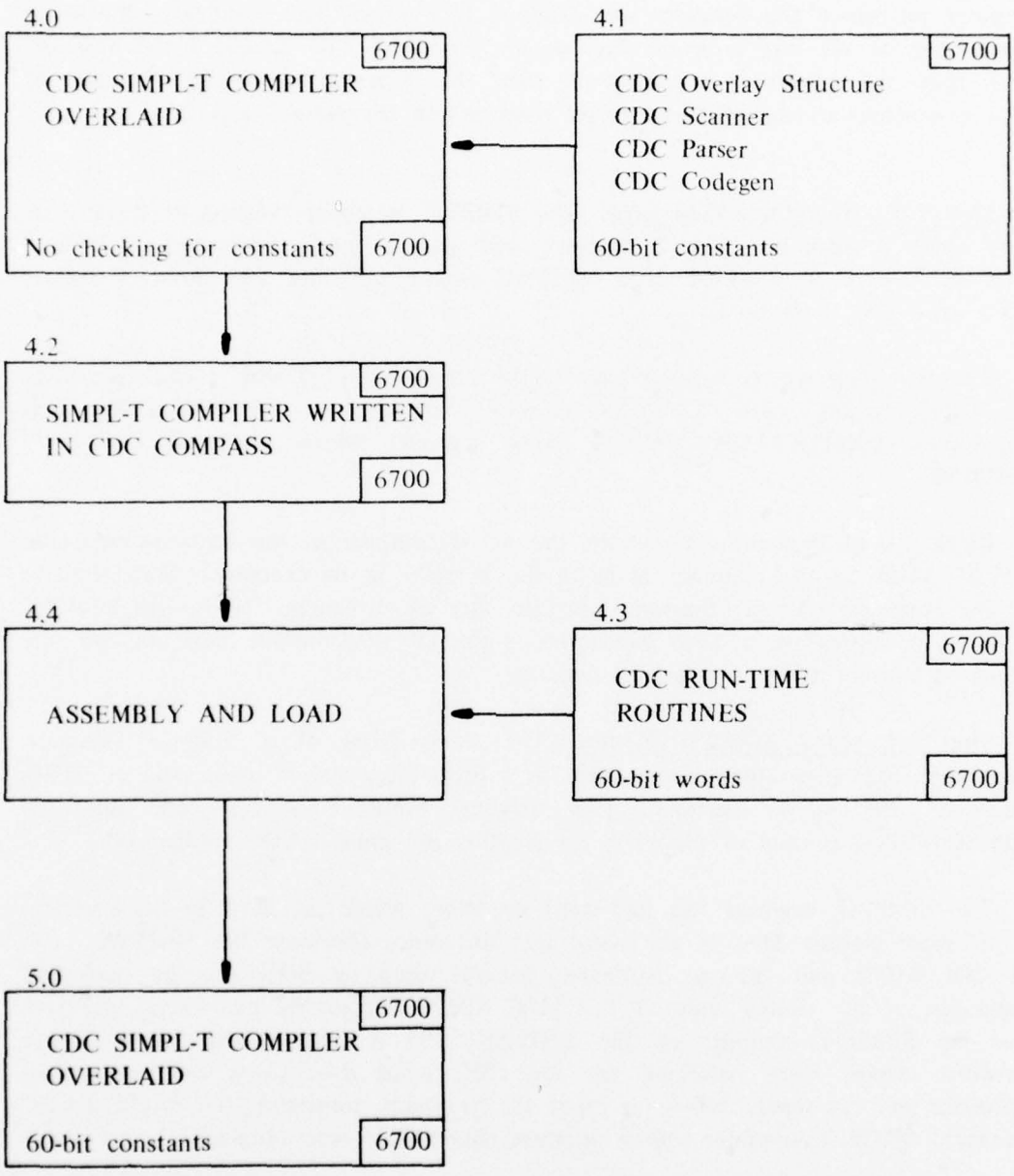


Diagram 7. SIMPL-T Compiler 5



Within each block, the number in the upper right corner indicates the specific computer on which the program was designed to execute. The number in the lower right corner of the box indicates the specific computer that produced the program. (Note that the notation is consistent with the superscript/subscript notation of earlier discussions of the host and target machines in Diagrams 1 and 2.)

**Step 1.0 – New Code Generator**. The SIMPL-T compiler consists of three basic passes: pass 1 (scanner), pass 2 (parser), and pass 3 (code generator). All three passes are written in SIMPL-T. The UNIVAC version can only have 36-bit constants as the word size is 36 bits.

The first step was to replace pass 3 (the code generator) with a code generator that would produce CDC COMPASS (assembly language for the CDC 6700). This generated a UNIVAC 1108 SIMPL-T cross compiler which produced CDC 6700 COMPASS.

Block 1.0 of Diagram 4 represents the initial compiler of the host machine (the UNIVAC 1108) as the compiler exists in the machine in its executable state (i.e., as machine language). As the diagram proceeds, any block assigned an integer label (1, 2, 3, etc.) represents a new executable stage of the compiler existing on the designated host or target in machine language.

**Step 1.1 – CDC SIMPL-T Source**. The three parts of a compiler (scanner, parser, and code generator) were needed in a format (part word addresses) especially suited to the CDC machine. The existing SIMPL-T source code for the UNIVAC 1108 was used as the basis for creating the needed CDC source code.

The SIMPL-T language has part-word operators which are used in the compiler for bit manipulation. Due to the word size difference (between the UNIVAC 1108 and CDC 6700) and the bit addressing scheme used by SIMPL-T, the part-word expressions of the source code of the 1108 SIMPL-T compiler had to be changed. Since the SIMPL-T compiler on the UNIVAC 1108 is written with macros, these part-word macros were redefined for the CDC word size. Using the new macro definitions and the macro source as input to the macro translator, the result, a CDC non-macro SIMPL-T compiler source program (Block 1.1), was obtained.

In block 1.1, the resulting source version of the compiler used only the right 36 bits of its larger 60-bit word and could accept only 36-bit constants. Actually, the COMPASS code generated was assembled into 60-bit words; but data (SIMPL-T constants) were, at this point, restricted to 36-bit words, sign extended.

Note in consulting Diagrams 4 through 7 that once on the target machine (the CDC 6700), successive modifications to the compiler were accomplished specifically to relieve the word size restriction. The final form of the compiler accepts 60-bit constants as data.

**Step 1.2 – SIMPL-T COMPILER Generated on the UNIVAC 1108 for the CDC 6700**. The CDC SIMPL-T source generated in step 1.1 was the input into the modified UNIVAC 1108 compiler of step 1.0. The resulting output was the SIMPL-T compiler written in CDC COMPASS (CDC assembly language). This version of the compiler may be run on the target machine. SIMPL-T is its input and it generates CDC COMPASS.

**Step 1.3 – CDC Run-Time Routines**. The necessary SIMPL-T compiler run-time support routines were written in CDC COMPASS. The run-time support routines include I/O, string, part-word, traceback, error handling, setup, and memory management routines (a complete list is given in Appendix A).

This is the beginning of Phase 2 (the CDC phase). The initiative now shifts to work on the target machine. All computing from this step on is accomplished on the CDC 6700, with no additional need for the host machine (exception noted later).

**Step 1.4 – Assembly and Load (on the CDC 6700)**. Block 1.4 illustrates the actual installation of the new compiler on the target machine (the CDC 6700) for the first time. Installation involved the integration of the compiler written in COMPASS (block 1.2) with the necessary runtime routines (block 1.3). The details of this procedure follow.

The initial installation of the SIMPL-T compiler on the target machine was a more complicated procedure than the installation of the successive versions of the compiler (blocks 2.4, 3.4, and 4.4). This installation (of block 1.4) was complicated by the fact that the SIMPL-T compiler written in CDC COMPASS (block 1.2), as generated by the UNIVAC 1108, was not in a format compatible for input into the CDC 6700. Before the COMPASS output from the UNIVAC 1108 (hereafter referred to as the COMPASS file) could be assembled on the CDC 6700, it had to be converted from UNIVAC internal character code (ASCII) to CDC internal character code (display). To facilitate the conversion, a program was written for the UNIVAC 1108 to read the COMPASS file and produce a tape reorganizing the file into an acceptable format for the CDC 6700.

The resulting tape was physically carried to the target machine. It represented the information of block 1.2 translated into a format acceptable to the CDC 6700. In this form, the COMPASS version of the SIMPL-T compiler (block 1.2) was assembled (block 1.4) to produce the relocatable CDC binary version of the SIMPL-T compiler.

An essential aspect of the assembly procedure is the integration of the compiler (1.2) with its necessary runtime support routines (1.3). In block 1.4, both were combined on the binary level to produce a functional SIMPL-T compiler for the CDC 6700 (block 2.0). The CDC SCOPE 3.3 operating system does not have a standard loader which allows a user library like the SIMPL-T run-time library. Therefore, the nonstandard LINK loader from New York University was used in this capacity.

**Step 2.0 – Non-Overlaid CDC SIMPL-T Compiler**. The first SIMPL-T compiler assembled on the target machine (block 2.0) was a non-overlaid version requiring about 55,000 60-bit words of storage. Diagram 5 traces the procedure involved in converting this compiler (block 2.0) to a fully overlaid version (block 3.0).

**Step 2.1 – CDC Overlay Structure**. The size (55K) of the non-overlaid version (block 2.0) required more than half of the available user memory (about 100K) on the CDC 6700. (Maximum memory size of the CDC 6700 computer is 131K words.) In the CDC SCOPE 3.3 multiprogramming, multiprocessing operating system turnaround time of a job is based upon a job's memory usage, its CPU utilization (execution time), and its external priority. The target site has elected to make memory usage the primary consideration in determining job turnaround time. Therefore, the SIMPL-T compiler in its non-overlaid version (a program requiring more than half of the available user memory) would command a low priority in the multiprogramming environment. Its turnaround time would be several hours, disallowing the use of interactive programming. For these reasons, a more efficient overlaid version of the compiler was sought.

Overlays in the CDC system must be called in as a part of the source program. Therefore, the source program (block 2.1) was the object of modifications to effect the desired overlay version of the compiler. Specifically, the main driving routine of the SIMPL-T compiler (written in SIMPL-T) (block 1.1) was rewritten (block 2.1) to call in the various desired overlays. The overlay structure used was similar to that used on the UNIVAC 1108 version. The scanner, parser, code generator, error handler, and cross reference generator each became an overlay call by the main driving routine of the compiler. The symbol table along with the main driving routine are in the root (resident) overlay.

**Step 2.2 – COMPASS for Overlay Version**. The new main driving routine with overlay calls was compiled by the non-overlaid SIMPL-T compiler. Note that this is the first time that a version of the SIMPL-T compiler has been used on the CDC 6700. The resulting output (block 2.2) is an overlaid COMPASS version of the SIMPL-T compiler written in COMPASS.

**Step 2.3 – Overlay Support Added**. A routine to interface SIMPL-T with the system overlay loader was added to the run-time support routines.

**Step 2.4 – Assembled and Load**. The COMPASS overlaid version was assembled and linked with the run-time routines.

**Step 3.0 – CDC SIMPL-T Compiler**. The SIMPL-T compiler produced by the step 2.4 required about 28K* core. This compiler could be run interactively, and batch turnaround time was below an hour. At this point (block 3.0), there existed a useful CDC SIMPL-T compiler. Only the lack of 60-bit constants remained a serious problem and the object of further modifications.

**Step 3.1**. As stated earlier, the SIMPL-T compiler checks the size of constants. The scanner portion of the SIMPL-T compiler written in SIMPL-T (blocks 1.1 and 2.1) had a check for 36-bit constants and the scanner itself used a 35-bit constant. This check was a carry-over from the UNIVAC 1108 version where it was necessary to insure that no constant would exceed the effective UNIVAC 1108 word size of 36 bits. However, on the CDC 6700, the greater word size allows for a 49-bit integer word and a 60-bit octal or binary word. The checks in the compiler that restricted the word size of constants to 36 bits were obsolete and inhibiting on the CDC machine. Changing the compiler's scanner to include a check procedure oriented to the CDC 6700, first required the generation of a compiler with no checking (4.0) which would allow for the assembly of a compiler with modified checking procedures (4.1).

Note that a modified compiler allowing a greater word length (60 bits) could not be assembled on the existing compiler because checking (block 3.0), as the mere introduction of the constant, would set the new limits (because said constant would exceed 36 bits) and would abort the compiling procedure. Therefore, the SIMPL-T version of the compiler was modified to allow no checking of constants (block 3.1).

*For size comparison only, CDC FORTRAN and COMPASS each take a minimum of 20K and COBOL about 24K.

**Step 3.2 – COMPASS Version With No Checking for Constants**. The SIMPL-T source, which would allow any size constants and did not contain any constants over 36 bits itself (block 3.1), was compiled by a previously generated compiler (block 3.0) and produced a COMPASS version (block 3.2).

**Step 3.3 – CDC Runtime**. No change was necessary.

**Step 3.4 – Assemble and Load**. The modified compiler (3.2) with no checking was installed on the CDC 6700, as were the run-time support routines (3.3).

**Step 4.0 – CDC SIMPL-T Compiler Without Checking of Constants**. This version of the compiler was only used to compile the succeeding version of the compiler which did checking for the CDC word length.

**Step 4.1 – Modify Source to Allow 60-Bit Constants**. The SIMPL-T version of the compiler was rewritten (block 4.1) to allow a checking procedure based on CDC's hardware which supports 49-bit integers (48 bits plus 1 bit for sign). The reinstated checking procedure allowed a 48-bit integer constant and 60-bit octal or binary constants.

**Step 4.2 – COMPASS with 60-Bit Constants**. All the necessary changes had been made to the SIMPL-T source (4.1) to have a complete overlaid version of a CDC SIMPL-T compiler which supported the full-word size of the CDC computer (5.0). This source (4.1) was compiled and produced the final COMPASS version of the compiler (4.2).

**Step 4.3 – CDC Runtime Routines**. No changes were necessary.

**Step 4.4 – Assemble and Load**. The final compass version was assembled and linked with the run-time routines.

**Step 5.0 – CDC SIMPL-T Compiler**. The final compiler was a self-contained system capable of recompiling (4.1) and enhancing the compiler. The bootstrapping had generated a total of five compilers for the end purpose of developing the CDC version of the SIMPL-T compiler in its final state. This final compiler was functional on the CDC 6700 computer with overlay capabilities and support for the target machine's greater word length.

## DIFFICULTIES

### Testing Problems

**Overview**. As expected, testing proved to be the most complex aspect of the project. Testing accounted for approximately 45 percent of the total project time. The testing would have been more efficient if better programs had been written for this purpose. In retrospect, better sample programs should have been designed to serve as tests and more time should have been allowed for this phase.

A bootstrapping project causes some subtle testing problems not found in other projects. The testing procedures must take into consideration the following constraints:

1. The run-time routines for the target machine are necessary in order to check out the target code generator (which is initially generated on the host machine).

2. The target compiler is needed (at some time or another) to check out the run-time routines.

If this sounds like the typical circular argument about chickens and eggs and first and last, then the reader has begun to appreciate the problem. If the code generator produces correct code the first time the code is executed on the target computer, and if at the same time the run-time routines (some are necessary in order to run any program on the target machine) are correct, then there is no problem. The project is complete and no testing is needed. Unfortunately, this is seldom the case.

As stated above, problems do occur. Because the coding is not perfect, there is the necessity for carefully structured testing procedures. The primary reason that the code generator and run-time routines must be correct (adequately tested) is that testing and correcting an error can require a great deal of time. Just determining what code (e.g., code generator, run-time routine, etc.) is causing a given problem can prove to be difficult. In this project, the fact that some of the test programs and parts of the SIMPL-T compiler (i.e., scanner and parser) were not familiar to the tester made the testing phase of paramount importance.

In order to give the reader the proper scope and depth of the testing phase, the events are described in their chronological sequence as attempted during this project. A standard approach was used for testing the code generator. However, the testing procedure used for the CDC run-time routines was more optimistic than the

standard approach (i.e., writing the run-time test routines in COMPASS). The optimistic approach assumed that the code produced would, in fact, be correct (or at least sufficient to make the initial testing possible). At the conclusion of this section, a discussion of other possible approaches will be presented.

Testing the Code Generator. A standard testing approach was used. About 40 SIMPL-T routines were written to test the code produced on the initial UNIVAC 1108 version of the modified code generator. The checking was done visually, and was very tedious. When an error was located, it frequently required the rerunning of the test cases. For this checking, a convenient CDC computer would have proved beneficial (as opposed to the 70-mile separation of computers in this project).

Testing the Run-time Routines. In order to check the CDC run-time routines (written in COMPASS), there must exist a calling routine (in this case, a program written specifically for testing purposes). This test program (calling routine) may either be written in SIMPL-T, compiled on the UNIVAC 1108 and run on the CDC 6700, or it may be written in COMPASS, assembled and run on the CDC 6700. Writing the test programs in SIMPL-T is the approach requiring the least effort.

A logical approach to testing was used in this project. The basic premise in this approach was that the untested code generator (only visually checked) on the UNIVAC 1108 must be capable of producing correct CDC COMPASS code. Once this premise is accepted, checking logic is as follows: each SIMPL-T test routine has the primary function to test a CDC run-time routine. If an error occurs, then it is logically assumed to be in the run-time routine. This approach minimizes the amount of time required to write test routines because they are written in SIMPL-T rather than COMPASS. An additional benefit is the fact that this method produces a set of test cases that can be used in testing modifications to this compiler and any other future bootstrapping projects involving this compiler (currently being used for bootstrapping SIMPL-T to the Nanodata QM-1). Also, this method allowed the tester to make use of test routines that were previously developed in the SIMPL-T language for the UNIVAC SIMPL-T compiler effort.

One disadvantage to this approach appears when an error is made in the code generator. In fact, an error in the code generator compounds the problem by requiring debugging both the code generator and the run-time routine at the same time. Another disadvantage in using this method (of particular significance in this project) is the difficulty in tailoring the test program to isolate a run-time routine error. This difficulty arises because the change of even one line of code in the

SIMPL-T program may require returning to the UNIVAC 1108 computer, making the change, recompiling, and then repeating the run on the CDC 6700. Once again, the distance between the host and target machine used in this project proved to be bothersome and time-consuming. The COMPASS version of the SIMPL-T program could be modified by using the CDC INTERCOM (interactive editor)[9] system program to change or add new COMPASS statements to the test program, because COMPASS rather than machine language was used.

If a standard approach had been selected (as was selected in the original UNIVAC 1108 bootstrapping project), COMPASS would have been used to write the testing programs. The advantage in this method of operation lies in the fact that COMPASS does not require the use of the SIMPL-T compiler; therefore, the run-time routine may be carefully tested before any compiler generated code is run. An additional advantage lies in the fact that work could be apportioned between groups or locations. For example, Group A could be working on the code generator on the host machine while Group B worked on run-time routines on the target machine.

The one serious drawback to this approach is the complexity of writing test cases in COMPASS instead of writing them in SIMPL-T. The division of work responsibility among groups would have the disadvantage of increasing the number of man-hours spent on the project. (Intercommunication, low-level programming, and some redundant effort would affect this condition.) The fact that the test programs would have only minimal potential in checking future efforts should also be considered a drawback to this programming approach.

The decision to write the test programs in SIMPL-T can be summarized in the following three justifications:

1. There was no redundancy in coding or designing test routines, therefore leading to greater efficiency. Routines could be used later. High level language could be used throughout debugging routines.

2. The target language being produced by the modified SIMPL-T code generator was COMPASS and could, therefore, be modified (using interactive computing) to correct a minor code generator error or could be changed to a slightly different test case.

3. Since one person was working on the project, there was no need to partition the project into two parts (one for the host and one for the target sites), as in the traditional approach.

Once the decision was made to use SIMPL-T, the following approach was used. A group of about 20 SIMPL-T test programs was written to check the CDC run-time routines. First the test cases were run on the 1108 SIMPL-T system and the answers were saved for further comparison. Then the programs were compiled on the modified code generator version and the resulting COMPASS was run on the CDC system. A visual comparison of the outputs (UNIVAC 1108 and CDC 6700) was made. Any differences were accounted for and corrected. The types of problems that could (and did) occur were: (1) error in the UNIVAC 1108 SIMPL-T system; (2) error in the code generator; (3) differences in word sizes; (4) differences in routine implementation due to imprecise SIMPL-T language definitions; (5) error in the CDC SIMPL-T runtime routine; and (6) error or failure (software or hardware) of either computer system.

**Testing the Combined CDC Compiler and the Run-Time Routines.** After all the CDC run-time routines were checked and corrected in the above manner, about 20 test programs (fortunately available from the UNIVAC 1108 SIMPL-T compiler effort) were first run on the UNIVAC 1108 system and compiled, and the COMPASS was run on the CDC systems. Again, any differences between the UNIVAC 1108 answers were resolved.

The final test program was the SIMPL-T compiler itself. The compiler consisted of about 6000 SIMPL-T statements and produced about 100,000 lines of COMPASS. The COMPASS was assembled and run. Four errors were found: two errors in the code generator and two in the run-time routines (see Error Analysis Section). Finding these errors required a great deal of time due to their complexity and lack of detailed knowledge of the scanner and parser. The two code generator problems were corrected by editing the COMPASS. Therefore, there was no need to recompile on the UNIVAC 1108 system. Once the two run-time routine errors were isolated and corrected, the compiler could recompile itself.

## Absolute Problems

In this project, as with any project, problems developed that were inherent in the project and therefore unavoidable. Although some time was spent in attempting to overcome these difficulties, some could not be resolved. This section will illustrate these absolute problems.

The major problem grew out of the unavoidable fact that the project required 60-bit word size arithmetic, and the UNIVAC was designed to handle a maximum of 36-bit word size arithmetic. The fixed-length word size-dependent arithmetic of

the UNIVAC necessitated the generation of two extra versions of the SIMPL-T compiler in order to accomplish the bootstrapping operation. A SNOBOL approach was originally believed to be the solution to this problem. However, although theoretically possible, SNOBOL proved (at least the available version) to also be limited to 36-bit word size arithmetic. Without an adequate 60-bit version, the only advantage offered by SNOBOL would have been in allowing the project to be bootstrapping using only the CDC 6700. This would in no way solve the word size dilemma. No other logical alternative (except radically modifying the SIMPL-T version of the UNIVAC 1108 compiler) presented itself.

An important aspect that the above arithmetic problem presented during this project was that it proved far more difficult to move from a smaller word size machine to a larger word size machine than it would have been to accomplish the bootstrapping from the larger to the smaller. Because the compiler was implemented to support integers only at the word size of the host machine, the target machine with a larger word size than the host compiler cannot allow the larger constants (for example a 59-bit constant could not be used on the UNIVAC 1108 SIMPL-T compiler).

The fact that the arithmetic of the computers was one's complement simplified the conversion effort. That is, the internal representation of integers up to the word size of the host machine (up to 36 bits) was bit for bit compatible with the CDC integer. However, if the target machine were two's complement, extra checking and conversion would have been needed to convert from one's complement to two's complement.

Character sets are traditionally a problem in bootstrapping. In this project, the SIMPL-T compiler on the UNIVAC 1108 used 8-bit ASCII internal code and the CDC 6700 used a 6-bit display code. The compiler only required two characters which did not exist in the 6-bit code, and the compiler did not use all of the display code characters. This permitted a mapping of the 8-bit ASCII code into the 6-bit display code. The only drawback with this character set mapping was that the SIMPL-T compiler assumed that ASCII was being used. For example, the scanner used an ASCII hash coded symbol table. This forced the CDC version of the compiler to go through an extra conversion (display code to ASCII).

The primitive CDC instruction repertoire is the basis for the third problem. For the most part, the computer must execute more instructions to implement an alogrithm on the CDC 6700 system than on any other computer of this class. For example, there are no character instructions and no indirect addressing on the CDC machine. There is no way to overcome this problem.

The final problem stemmed from the distance involved between the host and the target computers. Visual checking of the code produced by the UNIVAC 1108 was required, rather than checking by running test cases directly on the CDC machine. The most logical solution would be to establish a terminal to allow access to either one or both computers; however, the limited scope of this project made this solution impractical.

## Enhancements

The parser in the original version of the UNIVAC 1108 compiler was simplified. The parser required the code generator to be more complex than necessary due to the failure of the parser to identify on which side of the assign statement the part string/part word operators occurred. This deficiency of the parser required the CDC version of the code generator to save all of the part string/part word operators until the assignment statement was read.

Note that after the completion of the CDC code generator, this deficiency of the parser was corrected in the original UNIVAC 1108 compiler. Consequently, the modified code generator (as written before the change for this project) performs extra checks to compensate for a deficiency that no longer exists in the parser of the original compiler.

Knowledge of this type of temporary space is needed by the code generator. For simplicity, the parser was changed to indicate the type of temporary, as part of the internal language, used by the compiler. Otherwise, determination of the type of temporary would have come from consideration of the operands or operation code.

## Mendable Problems

As previously mentioned, the word size difference between the UNIVAC 1108 and the CDC 6700 computers caused problems. The SIMPL-T compiler was designed to be transportable to machines of smaller word size (including mini computer and ASCII-oriented machines). Hence, moving to a machine with a larger word size caused some special problems. Often the larger word size of the CDC was not fully utilized. For example, the SIMPL-T compiler symbol table used only the right 32 bits of the CDC's 60-bit word size capability. One way to improve on the word size would be by redefining the macros that access the compiler symbol table to use 48 of the 60 bits. Some investigations have been conducted along this line; and,

preliminary study has indicated that this method is feasible. However, problems seem likely to develop because not all routines use consistent access methods.

The part-word operators allow accessing of bit fields within a word. The SIMPL-T language defines the left-most bit (the sign bit) as bit number zero. This by itself would cause portability problems because a SIMPL-T program referencing bit zero would refer to an entirely different bit. For example, bit zero in the UNIVAC is $2^{35}$ and in the CDC bit zero is $2^{59}$. As stated earlier, macros were redefined in this project. A transportable notation having the bits numbered in the opposite manner would have been beneficial. For example, zero would be $2^{0}$ and bit 35 would be $2^{35}$. This would save time and improve portability.

## General Bootstrapping Problems

This section will deal with the problems that the writer has learned to expect in any bootstrapping procedure. The problems and observations involve the development of the program from the UNIVAC to the CDC.

### UNIVAC Problems.

1. Difficulty arose due to a lack of adequate documentation of the parser and scanner. The minimal documentation required an extra time expenditure in order to obtain the necessary basic understanding of these two parts of the compiler before writing the new code generator.

2. Work on the CDC code generator required interfacing with the other parts of the compiler (scanner and parser). The parser and scanner were being rewritten and improved at the same time the code generator was being written. Unfortunately, this caused some hindrance in the testing of the new code generator. (For example, some external and entry point names originally used in the scanner and parser were changed, causing the termination of the testing procedure of the code generator until the new names were discovered.)

3. An unanticipated overflow of the compiler tables necessitated a reorganization of the code generator into three separately compiled modules. This problem of reorganization could have been avoided if compiler table overflow had been foreseen.

CDC Problems.

1. Initially, the compiler would run on the CDC 6700 under the SCOPE 3.3 system and, later, on the SCOPE 3.4 system. Detailed knowledge of the CDC systems and computer hardware was needed. One of the more difficult problems that arose included designing a run-time environment that would be efficient and would interface with CDC COMPASS and FORTRAN.

2. Input/Output and run-time error recovery required precise knowledge of interfacing with the CDC operating system. More difficulty arose in analyzing the operating system at the technical level than in implementing the necessary interfacing.

## CDC RUN-TIME LIBRARY

CDC support required the most effort of any one area of the project. The major reason was the use of COMPASS as the programming language. COMPASS appeared to be the only reasonable language in which to write most of the run-time routines. Some of them could have been written in SIMPL-T, but this would have required that the compiler produce correct code before the run-time routine could be used. Perhaps if the project were to be done again, more time could be spent in investigating the language (especially modification of SIMPL-T) for run-time routines. A complete summary of the routines that were written is given in Appendix A. The following is a brief overview of the number of routines written: compiler intrinstic routines (33); UNIVAC system routines (2); and SIMPL-T program control, error handling, and basic I/O routines (27).

## IMPLEMENTATION RESTRICTIONS

Even though the writing of a basic nonoptimized code generator was chosen as the alternative to the more complicated optimized version, the rewriting and modification of the code generator for the SIMPL-T compiler were further simplified by the following restrictions:

1. Case statements were not limited in range in the UNIVAC 1108 version, causing the dilemma of whether to build a jump table or use a list. Given the alternative of limiting the case statement range or incorporating a

sophisticated decision making algorithm into the code generator, the decision was made to limit the case statement selector range from 0 to 99 in the CDC version. Therefore, a jump table was very easily implemented.

2. The SIMPL-T language has no restriction on the number of actual parameters. The UNIVAC 1108 code generators used a linked list within the newly available locations of the symbol table. The UNIVAC 1108 approach required routines for allocating and deallocating this space. In rewriting the code-generator, this approach was simplified by restricting the number of actual parameters (20). This restriction allowed the use of normal SIMPL-T recursive procedures coupled with a local array that contained the actual parameters for the appropriate call statement.

## STATISTICS OF PROJECT

### DEFINITIONS OF TERMS

Detailed timing information was kept on a daily basis for each component of the project. Difficulties occurred in trying to define discrete components (e.g., design, code, and checkout) and in being able to break down the time expenditures of each day into these categories. The terminology involved for this portion of the project is not standardized; therefore, the discrete categories decided upon are listed and defined below:

| | |
|---|---|
| DISCUSSION | Discussing global aspects of project and method used to develop the SIMPL-T UNIVAC 1108 compiler |
| BACKGROUND | Researching reference material; writing and running special purpose programs to display the internal workings of the compiler (on the UNIVAC 1108); getting the necessary background knowledge and studying source listings for the scanner and parser |
| DESIGN | Designing; flow charting; outlining |
| REDESIGN | Redesigning a part of the project because the initial design was incomplete |

| | |
|---|---|
| CODE | Writing on coding sheets; including some minor designing |
| CHECK | Visually checking logic of code; desk checking |
| DEBUG | Detecting, locating, isolating, and eliminating mistakes, malfunctions, or faults |
| TEST | Writing or running test cases; visually analyzing the results |
| KEYPUNCH | Keypunching or entering code into a terminal |
| SETUP | Preparing files, tapes, and jobs; creating bootstrap tapes on the UNIVAC 1108 and transporting them into the CDC file system |
| SYSTEM CHANGES | Making changes required by external operating system or compiler changes |
| SYSTEM PROBLEMS | Making corrections or redoing work as a result of a system malfunction (software or hardware) |

The project was separated into two parts; the UNIVAC phase and CDC phase. Each of these two phases was divided into the twelve different components previously defined. The components were also grouped together to give percentages for the general areas of (1) designing (discussion, background, design, and redesign), (2) coding (code), (3) testing (checking, debugging, and testing), and (4) miscellaneous (keypunch, setup, system changes, and system problems).

The figures below represent work on the project over 9 calendar months (150 days or 746 hr). Only pure work hours are considered; that is, only time spent on the project.

## UNIVAC PHASE OF PROJECT

The UNIVAC phase involved the general design of the code generator, the design of the CDC run-time environment, and the coding and check out on the UNIVAC 1108 of the CDC code generator. Table 1 contains the hours, percentages, and group totals for this first phase.

**Table 1. Part 1 – UNIVAC Phase of Project**

| | Hours | Percentage |
|---|---|---|
| Group 1 – DESIGNING | | |
| Discussion | 13 | 4 |
| Background | 43 | 12 |
| Design | 34 | 10 |
| Redesign | 18 | 5 |
| | 108 | 31 |
| Group 2 – CODING | | |
| Code | 59 | 17 |
| | 59 | 17 |
| Group 3 – TESTING | | |
| Check | 22 | 6 |
| Debug | 37 | 10 |
| Test | 80 | 23 |
| | 139 | 39 |
| Group 4 – MISCELLANEOUS | | |
| Punch | 17 | 5 |
| Setup | 8 | 2 |
| System Changes | 14 | 4 |
| System Problems | 8 | 2 |
| | 47 | 13 |
| TOTAL | 353 | 100 |

Note that the design percentage of the UNIVAC phase was kept to a minimum due to the use of a nonoptimized code generator. If the more sophisticated optimized code generator for the UNIVAC 1108 had been written, it would have required much more design time, perhaps 50 percent of the total work time on the UNIVAC Phase, as opposed to the actual 31 percent for this project.

### CDC PHASE OF PROJECT

The CDC phase involved the design, the coding, and the testing of many (about 50) fairly independent and normally small run-time routines. In addition, the final testing and bootstrapping of the compiler was done in this phase. Table 2 contains the hours, percentages, and group totals for this second phase of the project.

### ENTIRE PROJECT

Before looking at the figures for the project as a whole, some comparison and discussion should be made about the percentages for each of the two phases:

| Phase | Designing | Coding | Testing | Miscellaneous |
|---|---|---|---|---|
| 1 (UNIVAC) | 31 | 17 | 39 | 13 |
| 2 (CDC) | 14 | 29 | 50 | 7 |

The differences in the designing percentages are not surprising, as the second phase required very little time to design the single function, well-defined run-time routines.

Both the coding and testing for the CDC phase were greater than for the UNIVAC phase, due to the large number of small run-time routines that needed to be written and checked. Most of the difference in the miscellaneous group can be explained by the fact that professional keypunching services were not available at the UNIVAC site. Therefore, the programmer had to type the routines in at a terminal. However, this did allow on-the-spot corrections. Conversely, perhaps a little more coding time was taken in the CDC phase because of the use of coding forms and a more rigid format.

Table 3 has the combined figures for the project as a whole. Again, the designing percentage (22 percent) is probably a minimum figure. Also, 23 percent

Table 2. Part 2 – CDC Phase of Project

| | Hours | Percentage |
|---|---|---|
| Group 1 – DESIGNING | | |
| Discussion | 11 | 3 |
| Background | 33 | 8 |
| Design | 12 | 3 |
| Redesign | 0 | 0 |
| | 56 | 14 |
| Group 2 – CODING | | |
| Code | 114 | 29 |
| | 114 | 29 |
| Group 3 – TESTING | | |
| Check | 22 | 6 |
| Debug | 48 | 12 |
| Test | 125 | 32 |
| | 195 | 50 |
| Group 4 – MISCELLANEOUS | | |
| Punch | 0 | 0 |
| Setup | 24 | 6 |
| System Changes | 0 | 0 |
| System Problems | 4 | 1 |
| | 28 | 7 |
| TOTAL | 393 | 100 |

Table 3. Totals for Project

| | | Hours | Percentage |
|---|---|---|---|
| Group 1 – DESIGNING | | | |
| | Discussion | 24 | 3 |
| | Background | 76 | 10 |
| | Design | 46 | 6 |
| | Redesign | 18 | 3 |
| | | 164 | 22 |
| Group 2 – CODING | | | |
| | Code | 173 | 23 |
| | | 173 | 23 |
| Group 3 – TESTING | | | |
| | Check | 44 | 6 |
| | Debug | 85 | 11 |
| | Test | 205 | 28 |
| | | 334 | 45 |
| Group 4 – MISCELLANEOUS | | | |
| | Punch | 17 | 2 |
| | Setup | 32 | 4 |
| | System Changes | 14 | 2 |
| | System Problems | 12 | 2 |
| | | 75 | 10 |
| | TOTAL | 746 | 100 |

for coding is probably a maximum for a project of this type. The testing percentage is about right, and 10 percent for mundane tasks and outside problems is not unusual.

An interesting observation made in the light of the percentages of Table 3 is a consideration of how these percentages would have been effected if the more sophisticated, optimized code generator had been written for the UNIVAC phase. Note that the design percentage for this project was kept to a minimum, as stated above, due to the straight forward, simple nature of the code generator, run-time, and bootstrapping procedures chosen. However, if an optimized code generator that produced relocatable code were written for the UNIVAC phase, then the designing for the entire project would have approached 35 percent, with the other areas being changed to 15 percent for coding, 40 percent for testing, and 10 percent for miscellaneous.

## PRODUCTIVITY

Productivity is normally given in lines per man-year, sometimes words per man-year. Units like these are very ambiguous and, therefore, must be qualified. For this project, the following items should be considered:

1. The level of optimization desired. (Some code would have to be discarded before an optimized production system could be produced.)

2. The background of the implementor and programmer environment (timesharing, part-time, one person, etc.).

3. The language(s) being used and the type of problem being programmed.

4. The origin of the code. (For this program, all of the code was written from scratch, not borrowed from existing sources.)

5. The amount of formal specifications and documentation desired. (Here, minimum.)

Brooks[10] appears to draw the conclusion that programmers produce a fixed number of statements per year regardless of the language. On the surface, this statement appears invalid for this project (compare line 1 with line 3 in Table 4: 22 vs 70 statements/day); in fact, two considerations cloud the issue:

1. This project used a computer with a primitive instruction set (CDC 6700). In the CDC 6700, for example, three instructions are necessary to transfer data from one memory location to another.

2. The second part of this project was composed of less complicated tasks than the first part. For example, there were 60 small, self-contained routines written for part 2, as opposed to the one more complex code generator written for part 1.

**Table 4. Productivity**

| | Source Language | Statement Language | Statements | Lines | Words | Hours | Statements/ Hour | Statements/ Day |
|---|---|---|---|---|---|---|---|---|
| Part 1 | SIMPL-T | SIMPL-T | 1000* | 1700 | 9536 | 353 | 2.8 | 22 |
| | SIMPL-T | COMPASS | 18606 | 20007 | 9536 | 353 | 53.0 | 424 |
| Part 2 | COMPASS | COMPASS | 3500 | 5100 | 3836 | 393 | 8.8 | 70 |

Where:

Statements – Include only lines with executable statements (no comment lines, no declarations, and no initialization; and, includes only statements contained in the code statements) written for the code generator and CDC run-time routines (no statements from test cases).

Lines – Includes comments, declarations, etc.

Words – CDC 60-bit words (include data storage).

Hours – Total project hours – includes time spent on all aspects of the project and all 12 components considered earlier in the separate time estimates. (See Tables 1, 2, and 3).

*The SIMPL-T code generator produced COMPASS statements. These COMPASS statements (produced by the 1000 SIMPL-T source statements) are described by the second line of this table. The statistics here are a function of the quality of the code generator and the machine instruction set of the CDC 6700.

Another statement that Brooks makes (confirmed in this project) is: "Programming productivity may be increased as much as five times when a suitable high level language is used."[10] The comparison of line 2 in Table 4 (424 statements/day), which is the number of COMPASS statements produced per

day by the 1000 SIMPL-T statements, with line 3 (70 statements/day), programming directly in COMPASS, indicates a ratio of 6:1. Of course, other items must be considered in evaluating this ratio (e.g., quality of the code generator). Nevertheless, the use of a higher order language does greatly improve programming productivity. Brooks, when discussing productivity, mentions four studies: OS/360's data – 2000 to 3000 instructions/yr; Aron's data – 2500* instructions/yr; Harr's data – 2230 words/yr; and Corbato's data – 1200 PL/I lines/yr. For this project, the productivity was about 5000 SIMPL-T statements and about 17,500 COMPASS statements/yr. The project figures range from two to eight times that of the above studies. Therefore, with a lot of assumptions, one could estimate that structured programming might improve a programmer's productivity by about 100 percent.

## ACTUAL TIME VS ESTIMATED TIME

As the data in Table 5 indicate, the initial estimations for time were, on the average, about 17 percent low.

**Table 5. Actual Time vs Estimated Time**

| | Hours | | Percent |
|---|---|---|---|
| UNIVAC PHASE | Estimated | Actual | Difference |
| Background and Design | 160* | 108 | -32 |
| Code Generator | 160 | 245 | +53 |
| | 320 | 353 | +10 |
| CDC PHASE | | | |
| CDC Runtime Routines | 160 | 192 | +20 |
| Testing | 160 | 201 | +26 |
| | 320 | 393 | +23 |
| Total | 640 | 746 | +17 |

*One pure man-mo was the estimate.
1 man-mo = 8 hr × 20 days = 160 hr

*5000: some interactions (no testing)/2 = 2500 including testing.

**UNIVAC Phase, Background, and Design.** Researching and designing the modified code generator (for the UNIVAC 1108) took considerably less time than originally estimated (32 percent less time). The assumption was made that the interfacing of the new code generator would require a substantial effort since there was very little documentation to the contrary. In reality, the opposite proved true. The code used in the UNIVAC SIMPL-T compiler was straightforward and easily comprehensible. The approach of using CDC COMPASS in a nonoptimized code generator minimized the effort needed to complete the project.

**UNIVAC Phase, Code Generator.** The time spent in writing and in checking the code generator exceeded the original estimate by 53 percent. The coding step proved to occupy a relatively minimal amount of time, while the greatest amount of time was spent in checking the code produced by the code generator. Extra time was spent in an effort to improve this phase of the project. In spite of the procedures described below, a better approach for writing and checking the code generator could be developed.

One factor in the original underestimate was the physical location of the two computers. The 70-mi difference in machine location required a day's traveling time between assembling, running, and verifying the code produced. This procedure proved impractical.

An attempt was then made to minimize the errors in the code generator by "desk" checking or visually checking the code generator. This approach was only moderately successful. With only one person working on the program, it was highly possible to consistently overlook a logical error. Although a structured programming language was used (SIMPL-T), and no procedure was more than 50 lines of code (one exception), the programmer was "blind" to certain minor errors that would have been caught if the program were set aside for a period of time or examined by another person.

The primitive target machine caused the generation of a high number of machine statements for each high level SIMPL-T statement. Each of these had to be checked visually; and when corrections were made to the code generator, each test program had to be rerun. Since 40 test problems were used, this approach required an inordinate amount of man-hours.

An estimated five logic errors were detected during the visual checking of the code generator itself (written in SIMPL-T) and its output (COMPASS). Most of these errors occurred in special case situations (see the Error Analysis section for a further discussion.) Approximately 10 mundane errors occurred in 1000 statements.

**CDC Phase, CDC Run-Time Routines**. More run-time routines were needed than originally estimated, and debugging and error recovery routines were needed to aid in checkout of the run-time routines. An example of a routine added to the basic routines was "trace back." On an error, "trace back" lists information about the SIMPL-T routines that were invoked before the error occurred.

Originally overlooked was the necessity of making the CDC compiler more compatible with the CDC operating system. For example, overlay capability, dynamic file capability, and system communication enhanced the compiler.

This phase dealt solely with the designing and writing of the CDC run-time routines. All information concerning the use of the routines in providing testing for the code generator will be dealt with in the testing phase.

The designing and writing of the run-time routines took 20 percent more time than anticipated. Approximately 60 routines had to be written. They were fairly independent. A conventional technique (i.e., simple flow charts) and a more modern technique were both used to design routines.

Approximately 50 percent of the COMPASS run-time routines in this experiment were designed with each approach. In the more modern technique, even though COMPASS was being used, the program was first written (but not compiled) in a modified SIMPL-T. In other words, SIMPL-T was being used as a "design language"[11] or "description language."[12] It was found that the design language approach provided a more logical procedure than the conventional approach, as it allows the designer to concentrate at the functional level (control flow level) without having to worry about the implementation level.

Fifty errors occurred (10 of which were logical and 40 mundane) in writing the run-time routines. Only three logical errors and 15 mundane errors occurred when the more modern technique of first writing in a design language was used. These figures indicate that the likelihood of an error decreased with the design language approach. The amount of time required to write routines was approximately the same regardless of the method used.

**CDC Phase, Testing**. There was a +26-percent difference in actual testing time compared to the original projection. A detailed explanation of the reasons behind the overrun is included in the Testing Problems section of this report.

## ERROR ANALYSIS

This project involved three testing sections. The first testing was that of the code generator; 15 errors occurred in this phase. The second section tested was the CDC run-time routines; 50 errors occurred in this phase. The final testing involved the integrated project (installing the CDC version of SIMPL-T compiler as produced on the UNIVAC 1108 cross compiler); four errors arose in this phase. The following section will deal with explanations of the errors and possible solutions to the problem.

**Analysis of Code Generator Errors**. Five logical and 10 mundane errors occurred in this section. Most of the mundane errors were the result of the programmer using the wrong variables, although some mundane errors occurred when logical errors were corrected. The logical errors occurred from lack of adequate time spent on analyzing the design of the code generator. These logical errors occurred in fairly unique situations. For example, errors occurred when two of the many possible ways a formal parameter could be passed as an actual parameter were overlooked. Therefore, errors would have decreased if more time were spent in designing the code generator and if another person was available to check the design.

**Analysis of Run-Time Routine Errors**. A surprisingly large number of errors occurred in the run-time routines (40 mundane and 10 logical). Several of the mundane errors were of a bookkeeping nature (e.g., destroying the register, using the wrong register, forgetting a necessary operator, etc.). Most of the errors of this type are not unfamiliar to assembly language programming. Other mundane errors were due to incorrect instruction sequences. More familiarity with the hardware should decrease errors of this nature. The logical errors occurred in two areas: writing string package routines and the I/O area. Because the strings were stored 10 characters/word, the algorithm for handling the strings had to consider many boundary conditions. Some boundary conditions were overlooked. An attempt to minimize and to share code caused minor interconnection problems. The logical errors in the I/O section were due to a lack of understanding of the operating system I/O macros. Most of the problems in this section developed from lack of familiarity of programming at the assembly language level.

**Analysis of Integration Errors**. In spite of the fact that the first two sections of the project were checked out and thought to be correct, four errors (two in the code generator and two in the run-time routines) developed when the programs were integrated. The first error in the code generator was the result of an error in the algorithm for calculating the address of one element in a string array. This error

would only occur when involved with more than 10 characters in an element, and the element being initialized to a string of more than 10 characters. The second error involved passing an externally defined file name to a called routine. Both of these code generator errors could have been caught with better design of the code generator or more complete test routines.

The first run-time error occurred in the part string operator routine when a null-string should have been returned but was not returned by the run-time routine. This error was the result of poor documentation on treating nonfatal errors. None of the previous test cases had presented this problem. Once discovered, this error was easily repaired. The second error occurred in passing a string (by value), where the string was defined as a larger size than the current size. The run-time routines contained an error that would not allow the string to be expanded by the called routine. This particular error had not previously appeared in the test routines and, once isolated, was easily corrected.

An analysis of the error pattern points to the following observations:

1. Additional personnel should review the programs.

2. Better test cases should be written.

3. Logical errors usually occur in specialized situations.

4. Detailed knowledge of the system in use is a necessity.

5. Detailed language documentation which includes all special cases (e.g., error conditions) is essential.

**Summary of Errors.** Table 6 gives the errors involved in the testing of phases 1 and 2. There were also four errors found in the final integration testing of phases 1 and 2. It is interesting to note that the ratio of statements written to errors made (this includes only errors found, of course) appears to be relatively independent of the source language programmed in (i.e., there was one error/67 SIMPL-T statements and one error/70 COMPASS statements). If this can be shown to be consistent for individual programmers, one could show that the use of a high-order language reduces the words of storage generated per error ratio, just as it increases productivity (i.e., in this case, 636 words/error for SIMPL-T generated code vs 77 words/error for COMPASS generated code). Of course, this ratio cannot be taken too literally as a more efficient compiler should reduce the total number of words of COMPASS code generated by the CDC SIMPL-T compiler.

Table 6. Errors

| | Source Language | Statements | Words | Errors | Statement/ Errors | Word/ Errors |
|---|---|---|---|---|---|---|
| Phase 1 | SIMPL-T | 1000 | 9536 | 15 | 67 | 636 |
| Phase 2 | COMPASS | 3500 | 3836 | 50 | 70 | 77 |

## SUMMARY AND CONCLUSIONS

This project proves that a transportable compiler can be bootstrapped from one computer to another computer in a reasonable amount of time (4 to 5 man-months).

Selecting a nonoptimizing code generator and using assembly language for the target language can minimize the time required on the host computer. Assembly language also allows more flexibility in debugging the initial modified code generator on both the host and target computers.

Time estimates as described by Brooks[10] state that coding should require 17 percent, design about 33 percent, and testing and integration 50 percent of the total project time. This project, with designing and coding requiring about 23 percent of the project time and testing requiring about 45 percent, would seem to support these findings if the following two items are considered:

1. This project required a minimum amount of design time.

2. It required a significant amount of coding in comparison to the design time.

Mills[12] has developed an approach that devotes 5 to 10 percent of the work time for coding, a large percentage in designing, a large percentage in "desk" checking (walk-throughs), and only a small percentage for testing. He maintains that a group of three people should visually verify the code to be correct before testing on the computer. This concept was applied by visually checking the SIMPL-T source for the code generator in this project. It was found that extensive visual checking for correct code did expedite the project. However, since only one person was checking the code, more errors occurred than should with a group of three. The visual checking approach should definitely be given more study.

The use of structured programming aided the programming effort by allowing a more modular and refined approach than had previously been found in languages like FORTRAN. Productivity was improved by an estimated 100 percent by using a structured approach. Also, using a high-order language improves productivity by about 600 percent. Describing the program in a programming description language appears to aid the programmer in conceptualizing the problem before he has to be concerned with the actual implementation details.

Brooks[10] points out that estimates made by programmers tended to be off by 100 percent. The fact that this project was only 17 percent off can be explained by the fact that most projects are not as clearly defined as was this one.

## REFERENCES


1. D. T. Ross, "The AED Approach to Generalized Computer-Aided Design," Proceedings of the ACM 22nd National Conference, 1967, pp 367-385.

2. W. M. McKeeman, J. J. Horning, and D. B. Nortman, ***A Compiler Generator***, Prentice Hall, NY, 1970.

3. V. R. Basili and A. J. Turner, ***A Structured Programming Language***, CN-14.2, University of Maryland Computer Science Center, College Park, MD, August 1975.

4. V. R. Basili and A. J. Turner, "A Transportable Extendable Compiler," ***Software – Practice and Experience***, 5, 1975.

5. V. R. Basili, "The SIMPL Family of Programming Languages and Compilers," TR-305, University of Maryland Computer Science Center, College Park, MD, June 1974.

6. University of Maryland Computer Science Center, SIN-41, College Park, MD, 1974.

7. J. McHugh and V. R. Basili, "SIMPL-R and Its Application to Large, Sparse Matrix Problem," *TR-310, University of Maryland Computer Science Center*, College Park, MD, July 1974.

8. W. Rheinboldt, V. Basili, and C. Mestenyi, "On A Programming Language for Graph Algorithms," BIT 12, 1972.

9. Control Data Corporation, ***6000 INTERCOM Reference Manual***, Version 4, Publication No. 60307100, Sunnyvale, CA, 1974.

10. F. P. Brooks, ***The Mythical Man-Month***, Addison-Wesley Publishing Company, Reading, MA, 1975.

11. Y. Chu, "A Methodology for Software Engineering," ***Software Engineering***, Volume SE-1, No. 3, IEEE Computer Society, NY, September 1975.

12. H. D. Mills, "On the Development of Large Reliable Programs," IEEE Symposium Computer Reliability, 1973, pp 155-159.

## BIBLIOGRAPHY


A. V. Aho and J. D. Ullman, ***The Theory of Parsing, Translation and Compiling***, Prentice-Hall, Volume 1, 1972, Volume 2, NY, 1973.

V. R. Basili and A. J. Turner, "Iterative Enchancement: A Practical Technique for Software Development," Precedings of the 1st National Conference on Software Engineering, IEEE, NY, 1975.

V. R. Basili, "SIMPL-X, A Language for Writing Structured Programs," TR-223, University of Maryland Computer Science Center, College Park, MD, January 1973.

H. Caine and E. K. Gordon, "PDL – A Tool for Software Design," National Computer Conference 1975, pp 271-276.

Control Data Corporation, ***6000 Computer Series FORTRAN EXTENDED VERSION 4***, Reference Manual, Publication No. 60305600, Sunnyvale, CA, 1974.

Control Data Corporation, ***6000 Computer Systems SCOPE Reference Manual***, 6000 Version 3.4, Publication No. 60307200, Minneapolis, MN, 1974.

Control Data Corporation, ***SCOPE 3.3 System Programmers***, Reference Manual, Publication No. 60306800, Minneapolis, MN, 1971.

Control Data Corporation, ***CYBERNET SCOPE 3.3***, Reference Manual, Publication No. 8661800, Minneapolis, MN, 1973.

Control Data Corporation, ***SCOPE 3.3 Analysis***, Publication No. 60298800, Minneapolis, MN, 1970.

Control Data Corporation, ***6400/6500/6600 Computer Systems***, Reference Manual, Publication No. 60100000, St. Paul, MN, 1969.

D. Gries, ***Compiler Construction for Digital Computers***, John Wiley and Sons, Inc., NY, 1971.

D. E. Knuth, ***The Art of Computer Programming***, Addison-Wesley Publishing Company, Reading, MA, 1969.

G. Leach and H. Golde, "Bootstrapping XPL to an XDS Sigma 5 Computer," ***Software – Practice and Experience***, 3, 557-244, 1973.

M. Richards, "BCPL: a tool for compiler writing and system programming," APIPS Proceedings, 34, 557-566 (SJCC 1969).

J. G. Perry, ***CDC SIMPL-T***, Naval Surface Weapons Center/Dahlgren Laboratory, Dahlgren, VA, Unpublished Report, 1976.

J. G. Perry, ***CDC SIMPL-T Internal Compiler Design***, Naval Surface Weapons Center/Dahlgren Laboratory, Dahlgren, VA, Unpublished Report, 1976.

B. W. Pollack, ***Compiler Techniques***, Averback Publishers, Inc., 1972.

SOFTECH Inc., ***AED User's Guide***, CDC 6000 Edition, Waltham, MA, 1973.

W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structural Design," IBM Systems Journal, Volume 13, No. 2.

A. J. Turner and V. R. Basili, "Notes on the Internals of the SIMPL-T Compiler," University of Maryland, College Park, MD, Unpublished Report, 1975.

J. A. Verson and R. E. Noonan, "A High-Level Macro Processor," TR-297, University of Maryland Computer Science Center, College Park, MD, March 1974.

M. H. Weik, ***Standard Dictionary of Computers and Information Processing***, Hayden Book Company, Inc., NY, 1970.

N. Wirth, "On PASCAL, Code Generation, and the CDC 6000 Computer," Stanford University, Stanford, CA, February 1972.

N. Wirth, "Program Development by Stepwise Refinement," COMM ACM 14, p. 221-227, April 1971.

# APPENDIX A

# CDC RUN-TIME SUPPORT ROUTINES

## INTRINSIC ROUTINES

| | |
|---|---|
| ABORT | ABORT SIMPL-T |
| CHARF | Convert to character |
| CHARVAL | Convert integer to ASCII character |
| DIGIT | Test Character for 0 to 9 |
| DIGITS | Test string for all 0 to 9 |
| ENDFILE | Write end of file |
| EOI* | Test for end of file (Read) |
| EOIC | Test for end of file (ReadC) |
| EOIF | Test for end of file (ReadF) |
| INTVAL | ASCII integer value of character |
| INTF | Convert to INT |
| LENGTH | Length of string |
| LETTER | Test for character A to Z |
| LETTERS | Test string for all A to Z |
| MATCH | Match a string to another |
| PACK | Character array to string |
| READ* | Free format read |
| READC | Read a card |

*Not supported

| | |
|---|---|
| READF | Read a file |
| REWIND | Rewind file |
| STRINGF | Convert to string |
| TRIM | Discard trailing blanks |
| UNPACK | String to character array |
| WRITE | Free format write |
| WRITEF | Write file |
| WRITEL | Write line |

## OPERATOR ROUTINES

| | |
|---|---|
| PARTOP$ | Part word operator |
| STASSG$ | String assignment |
| STBREL$ | String relationals |
| STCONC$ | String concatenate |
| STSUBS$ | Substring operator |

## RUN-TIME ENVIRONMENT SUPPORT ROUTINES

| | |
|---|---|
| GETMEM$ | Get more memory |
| PROCEN$ | Procedure entry |
| PROCEX$ | Procedure exit |

| | |
|---|---|
| STARUP$ | Start up SIMPL-T program |
| STALOC$ | Stack allocator |
| STCOPY$ | Copy string to stack |
| STFREE$ | Free string temporary |

## OPERATING SYSTEM INTERFACE ROUTINES

| | |
|---|---|
| CLOCK | Return clock time |
| DATE | Return date |
| CLOSE$$ | Close file |
| DEMAND$ | Return 0 if BATCH job |
| FLUSHB$ | Flush all BUFFERS |
| MTRREQ$ | Request monitor function |
| OPENF$$ | Open file |
| OVRLAY$ | Call in overlay |
| Q8NTRY | Initialize file tables |
| READC$$ | Read a card (one BUFFER) |
| READF$$ | Read a file (one BUFFER) |
| REMARK$ | Write to dayfile |
| RETURN$ | Release file |
| TIME | Return execution time |

| | |
|---|---|
| WRITEC$ | Write coded file |
| WRITEF$ | Write file (BINARY) |
| WRITEL$ | Write line to output file |

## INTERNAL CONVERSION ROUTINES

| | |
|---|---|
| CHAOUT$ | Character array to output file |
| INTCHA$ | Integer to character array |
| OCT20$$ | Convert word to 20 octal characters |

## ERROR HANDLING AND DEBUGGING

| | |
|---|---|
| ABORTM$ | Write message and ABORT job |
| ERALOC$ | Array location error |
| ERCASE$ | Case statement error |
| RPRIVE$ | Error recovery |
| TRBACK$ | Trace back |

## UNIVAC SYSTEM ROUTINES USED IN THE COMPILER

| | |
|---|---|
| $$SIGNON | Date and clock time |
| $$TIMER | Elapse time |

## ADDITIONAL I/O ROUTINES

| | |
|---|---|
| EOICF | End of coded file (READCF) |
| READCF | Read coded file |
| WRDATA | Write to output |
| WREND | Write end of file on output |